Cost-Effective Speculative Scheduling in High Performance Processors

Arthur Perais* André Seznec* Pierre Michaud* Andreas Sembrant† Erik Hagersten†
*IRISA/INRIA †Uppsala University
{arthur.perais,andre.seznec,pierre.michaud}@inria.fr
{andreas.sembrant,erik.hagersten}@it.uu.se

Abstract

To maximize performance, out-of-order execution processors sometimes issue instructions without having the guarantee that operands will be available in time; e.g. loads are typically assumed to hit in the L1 cache and dependent instructions are issued accordingly. This form of speculation – that we refer to as speculative scheduling – has been used for two decades in real processors, but has received little attention from the research community.

In particular, as pipeline depth grows, and the distance between the Issue and the Execute stages increases, it becomes critical to issue instructions dependent on variable-latency instructions as soon as possible rather than wait for the actual cycle at which the result becomes available. Unfortunately, due to the uncertain nature of speculative scheduling, the scheduler may wrongly issue an instruction that will not have its source(s) available on the bypass network when it reaches the Execute stage. In that event, the instruction is canceled and replayed, potentially impairing performance and increasing energy consumption.

In this work, we do not present a new replay mechanism. Rather, we focus on ways to reduce the number of replays that are agnostic of the replay scheme. First, we propose an easily implementable, low-cost solution to reduce the number of replays caused by L1 bank conflicts. Schedule shifting always assumes that, given a dual-load issue capacity, the second load issued in a given cycle will be delayed because of a bank conflict. Its dependents are thus always issued with the corresponding delay. Second, we also improve on existing L1 hit/miss prediction schemes by taking into account instruction criticality. That is, for some criterion of criticality and for loads whose hit/miss behavior is hard to predict, we show that it is more cost-effective to stall dependents if the load is not predicted critical.

In total, in our experiments assuming a 4-cycle issue-to-execute delay, we found that the majority of instruction replays caused by L1 data cache banks conflicts – 78.0% – and L1 hit mispredictions – 96.5% – can be avoided, thus leading to a 3.4% performance gain and a 13.4% decrease in the number of issued instructions, over a baseline pipeline with speculative scheduling.

ISCA'15, June 13-17, 2015, Portland, OR USA

ACM 978-1-4503-3402-0/15/06...$15.00
http://dx.doi.org/10.1145/2749469.2749470

1. Introduction

In out-of-order execution processors, the scheduler must keep instructions flowing through the pipeline at the highest rate possible, while dealing with hardware resource constraints and data dependencies. However, when an instruction I_0 is issued, several cycles are needed before the associated operation is effectively executed. The instruction is first marked as executable. Then, its register operands are read from the register file (PRF), then the effective operands are selected from the ones flowing out from the PRF and the ones coming from the bypass network. Finally, the operation is executed.

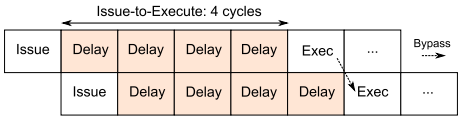


Figure 1: Pipeline diagram of two dependent instructions issued back-to-back in the presence of a 4-cycle delay between *Issue* and *Execute*.

To possibly execute a dependent instruction I_1 in the cycle following the execution of I_0 (back-to-back), the decision to issue instruction I_1 has to be taken before the effective execution of I_0. If the execution stage is N cycles after the issue stage, and instruction I_0 has latency P, then the dependent instruction I_1 must be issued at cycle P to get its operand at cycle $N+P$. This is illustrated in Fig. 1. In the remainder of this paper, we refer to $N-1$ as the *issue-to-execute* delay, e.g. if this delay is 4 cycles, I_0 will execute in cycle 5 while having been issued in cycle 0. We also refer to the action of speculatively issuing instructions without knowing the effective validity of their operands as *speculative scheduling*.[1]

Implementing a scheduler with back-to-back execution is a challenge even if the *Execute* and *Issue* stages are only one cycle apart [12, 20, 26]. Another difficulty lies in the

[1]Not to be mistaken for the *speculative scheduling* of Stark et al. which relates to the atomicity of Wakeup & Select in the scheduler [26].

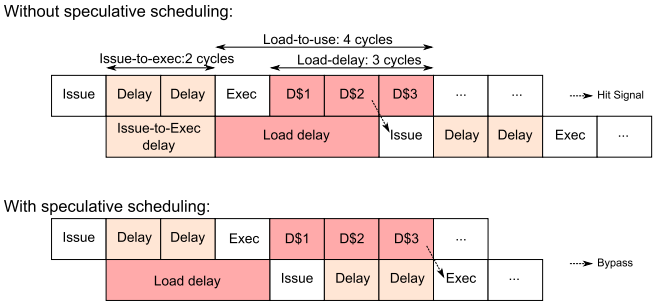


Figure 2: Pipeline diagram of a load followed by a dependent instruction. In the first case, the dependent is scheduled once the load is known to hit. In the second case, it is scheduled aggressively assuming there will be a hit.

decision to issue instructions *early* in the presence of an *issue-to-execute* delay.

Indeed, for some instructions or in some circumstances, one cannot guarantee the effective latency of an instruction. We illustrate this on a load from memory. The load latency may vary from the L1 cache response time – typically 1 to 3 cycles – to several hundreds of cycles if the last level cache misses. Decision to schedule dependent instructions after a load should then be delayed until knowing the availability of the hit signal of the load.

As illustrated in the top part of Fig. 2,[2] such a late decision would result in a large delay between the load and its dependent instructions. Therefore a load is generally assumed to hit in the L1 cache – since it is the most frequent case – and dependent instructions are issued accordingly, as depicted in the bottom of Fig. 2. Dependents of the dependents are also issued speculatively, and this continues until the hit/miss information is available. The whole dependency chain has to be canceled and replayed when a miss is encountered .

Multiported L1 caches are often physically implemented as banked caches. Two simultaneous accesses to the same bank (bank conflicts) will delay one of the accesses by one cycle. This represents another source of variable execution time that may cause replays.

Yet, even though both L1 misses and L1 bank conflicts lead to replays, the performance cost they entail is not similar. In particular, an isolated replay due to a bank conflict is costly because dependents will be delayed by the whole *issue-to-execute* latency due to their cancelation. Conversely, the latency of an L1 miss dominates the *issue-to-execute* latency. Therefore, less performance is lost when an L1 miss triggers a replay. In that latter case, loss comes from resource contention or replayed instructions that were independent of the miss, but the dependency chain of the load is not directly lengthened because of the misspeculation. Replays cost energy in both cases.

Nonetheless, the impact of these events on performance and overall misspeculation recovery cost is particularly high. For instance with a 4-cycle *issue-to-execute* delay, L1 data bank conflicts lead to a 4.7% average (gmean) performance loss compared with an ideal dual-ported data cache in our experiments. In this case, μ-ops replayed because of conflicts represent 5.1% of the issued μ-ops.

To reduce this impact, one can act on the replay mechanism and address the *effect(s)* of replays, or propose a scheduling mechanism to limit the number of scheduling misspeculations and tackle the *cause(s)* of replays. In this paper, we focus on the latter approach. Therefore, our propositions aim to be agnostic of the replay mechanism.

First, to address L1 data cache bank conflicts, we propose *schedule shifting*. Given a dual-load issue capacity and two loads issued the same cycle, the dependents on the second load are always issued with a one-cycle slack. In the case of a bank conflict, the second load will be penalized with a cycle. Hence, the extra cycle generated by the possible bank conflict can be tolerated. Although this lengthens the dependency chain of the second load, scheduling misspeculation associated with cache bank conflicts nearly vanish. *schedule shifting* is a low complexity solution, yet it allows to recover 2.8% out of the 4.7% performance lost due to L1 cache banking, on average. The gain stems from a decrease of 74.8% in the number of instructions replayed due to L1 bank conflicts.

Second, we show that the number of replays associated with L1 misses can significantly be reduced with a very simple L1 hit/miss filter. Our approach results in a 65.0% reduction of instructions replayed because of L1 hit/miss misspeculation, with performance similar to a baseline speculative scheme assuming that loads always hit.

Lastly, by combining these two approaches and for some criterion of criticality, we show that it is more cost-effective to delay dependents until hit/miss information is available rather than aggressively scheduling dependents on non-critical loads. Naturally, non-critical loads whose hit/miss behavior is easy to predict are allowed to wake up their dependents early. This simple scheme allows us to further reduce the number of replayed instructions: overall, 90.6% can be avoided, leading to a performance increase of 3.4% over a baseline speculative scheduling scheme.

The remainder of the paper is organized as follows: Section 2 summarizes related work and disclosed state-of-the-art while Section 3 gives an overview of the simulation framework we consider. Section 4 describes *Speculative Scheduling* in more depth by detailing well-known replay causes. Section 5 describes the impact of replays on performance as well as our mitigating techniques. Finally, Section 6 provides some concluding remarks.

2. Related Work

2.1. Replay and Selective Replay

When speculative scheduling is used, a replay mechanism is needed to handle wrong schedules.

[2]Throughout the paper, we assume that the hit/miss information is available one cycle before the data returns from the memory hierarchy.

Since the publication of Kim and Lipasti's work on scheduling replay in 2004 [12], state-of-the-art as disclosed by the industry *has not changed*. Thus, the only documented *actual* implementations are those of the Alpha 21264 [11] and the Pentium 4 [9]. They respectively illustrate two of the three main possible ways to handle schedule misspeculations: *replay* and *selective replay*. The third one is simply *refetch*, where the schedule misspeculation is treated as a branch misprediction, and instructions are removed from the pipeline then re-fetched. This last option is clearly costly from a performance standpoint and could only be implemented if scheduling misspeculations were very rare.

2.1.1. Selective Replay *à la* Pentium 4 As stated by its corresponding whitepaper [9], the Pentium 4 speculatively schedules dependents assuming that a load will hit in the L1 "in most cases". If the load misses, the processor is able to replay instructions that are bound to execute with incorrect source values selectively. No further details are given regarding the hit/miss predictor.

A possible implementation can be found in patents granted to Intel [15, 16, 17]. It allows instructions to be replayed via two means: a *replay loop* [17], and a *replay queue* [15].

The *replay loop* has a fixed latency, meaning that if an instruction is inserted in the loop to be replayed, it will be rescheduled after a fixed number of cycles. This can be used on an L1 miss but expected L2 hit, for instance. Depending on the particular microarchitecture, there can be several replay loops with different latencies [17] to accommodate the different events that can lead to a replay (L1 miss, DTLB miss, etc.).

Conversely, the *replay queue* can keep instructions to be replayed as long as necessary until they become ready (e.g. a signal coming from the cache). Instructions likely to wait for a long time are put in the *replay queue* instead of looping several times in the replay loop(s).

The choice of whether to put an instruction in the *replay loop* or in the *replay queue* is made by the *checker*. This piece of hardware is responsible for detecting that an instruction has executed incorrectly (e.g. L1 miss for a load that was assumed to hit). According to [17], there should be as many checkers as there are replay loops. Implemented in this fashion, *selective replay* has the good property that issued instructions immediately release their entry in the scheduler.

2.1.2. Replay *à la* Alpha 21264 The Alpha 21264 implements a more conservative replay mechanism [11]. Here, the processor tracks all the instructions that are in *the shadow* of a load. That is, all the instructions that have been issued at least *load-to-use* cycles after the load and that are inflight between the *Issue* stage and the *Execute* stage.

If the load misses in the L1 when it was expected to hit, then *all* those *shadowed* instructions are canceled and must be replayed, including independent ones. It is a much simpler mechanism than that of the Pentium 4: In the 21264, two pipeline stages have to be squashed and dependencies have to be restored for those squashed cycles.

According to [11], *"The queue* [Scheduler] *is **collapsing** – an entry becomes immediately available once the instruction issues or is squashed due to mis-speculation"*. Therefore, on a schedule misprediction, *"...all integer instructions that issued during those two cycles are **pulled back** into the issue queue to be re-issued later."*. It is not clear if instructions are pulled-back from the ROB or from a dedicated buffer. However, in the latter case, the parallel would have to be made with the *replay queue* of the Pentium 4.

2.1.3. A Few Academic Proposals Kim and Lipasti detail existing scheduling replay mechanisms in [12]. They shed light on the fact that efficient selective replay is hard, because the scheduler should stop selecting instructions whose result will not be available as soon as possible, while keeping on issuing instructions independent on the misspeculation. They also argue that selective replay is limited in scalability as dependency tracking requires more and more hardware as the machine width and depth grow. To remedy this, they propose *Token-based Selective Replay* where dependents on instructions likely to be mischeduled are replayed selectively while dependents on instructions unlikely to be mischeduled are squashed and re-inserted in the scheduler from the ROB.

Morancho et al. [19] study the impact of keeping instructions in the scheduler upon issue since they may need to be replayed. They also propose a scheme whereby issuing instructions are removed from the scheduler and put in a *Recovery Buffer*. The buffer has three separate arrays used for respectively: latency-predicted instructions not yet validated, instructions dependent on an instruction in the first array, and instructions needing to be replayed. This scheme accommodates both selective and non-selective replay. From a higher level, it resembles the *replay queue* described in Intel patent [15], although in Morancho et al.'s case, all issued instructions are put in the *Recovery Buffer* at issue time.

Ernst et al. propose the Cyclone scheduler [6] in which the number of cycles an instruction will wait for its operands is predicted before it is inserted in a queue. Each cycle, instructions in the queue advance toward the execution unit, thus, the queuing enforces the predicted delay before operand readiness. As Cyclone always schedules instructions speculatively, a replay mechanism is implemented by using scoreboarding logic at the functional units. That is, when an instruction exits the queue, it checks if its operands are ready. If not, the time at which they will become available is predicted again, and the instruction is re-inserted in the queue. Therefore, Cyclone implements a selective replay mechanism that shares some similarities with the *replay loop* [16] of Intel, except the latency is fixed in the latter case.

2.2. Latency Prediction

Ernst et al. employ latency prediction in their Cyclone scheduler [6], which is very similar to the dataflow prescheduler of Michaud and Seznec [18]. Instruction latency is predicted by accessing a table with the logical identifier of source registers.

This table contains the remaining number of cycles before the register becomes available. By applying the MAX function to each source, the latency can be computed. However, no particular mechanism appears to be implemented to handle variable latency instructions such as loads.

Liu et al. resort to address prediction for their load latency predictor [13]. They combine a *Latency History Table* (LHT) with a more complex cache latency predictor. The former is good at predicting loads that always exhibit the same latency. The latter is accessed using the address predicted for the load, and takes into account current accesses being made to the cache. It is able to detect that the load aliases with an older load whose L1 miss is being serviced. The latency of the older load will hide part of the latency of the younger. If no load to the same address is in flight, partial cache block information is used to determine if the load will hit or miss. Precise latency is required as their architecture sorts instructions by execution latency before sending them to the scheduler.

Memik et al. use a similar load latency scheme [14] where an address predictor is probed using the load PC. Then, using the predicted address, the Previously-Accessed Table is accessed to check for in-flight accesses to the same address. If there is a match, another instruction is currently accessing the data and can potentially hide the current load latency. If no match is found, the Cache Miss Detection Engine (CMDE) is accessed, still using the predicted address. There are as many CMDEs as there are cache levels, and each makes use of some partial cache tag information. When a CMDE predicts a miss, it is a *sure-miss*, however, if it predicts a hit, then it is really a *maybe-hit*. This scheme also aims to predict the precise latency of load instructions.

Finally, Yoaz et al. review existing memory dependence prediction techniques and propose novel schemes to predict the behavior of loads with regard to the cache hierarchy [29]. They focus on both Hit/Miss prediction and Bank prediction. They note that Hit/Miss prediction is binary, as Branch Prediction, therefore similar techniques can be used. Yet, some information cannot be kept precisely for Hit/Miss prediction: speculative behavior of previous accesses is not always known at execution time. Thus, history-based hit/miss prediction should be less efficient. The study also considers bank prediction for 2 banks using different sources of information (bank-history, control-flow, load-target, address prediction) and finds that the best performing scheme is actually the address predictor of Bekerman et al. [2].

3. Evaluation Framework

3.1. Simulator

In our experiments, we use the *gem5* cycle-level simulator [3] implementing the x86_64 ISA.

Table 1 describes our model: A relatively aggressive 4GHz,

Front End	L1I 8-way 32KB, 1 cycle, Perfect TLB; 32B fetch buffer (two 16-byte blocks each cycle, potentially over one taken branch) w/ 8-wide fetch, 8-wide decode, 8-wide rename; TAGE 1+12 components 15K-entry total ($\simeq$ 32KB) [23], 20 cycles min. branch mis. penalty; 2-way 8K-entry BTB, 32-entry RAS.
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ, 256/256 INT/FP register files; 1K-SSID/LFST Store Sets [5]; 6-issue, 4ALU(1c), 1MulDiv(3c/25c*), 2FP(3c), 2FPMulDiv(5c/10c*), 2Ld/Str, 1Str; Full bypass; ∞-wide WB, 8-wide retire.
Caches	L1D 8-way 32KB **Ports: 2R/2W**, 4 cycles load-to-use, 64 MSHRs; Unified L2 16-way 1MB, 13 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8; All caches have 64B lines and LRU replacement.
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Across an 8B bus; Min. Read Lat.: $\simeq$ 75 cycles, Max. $\simeq$ 185 cycles.

Table 1: Simulator configuration overview. *not pipelined.

Program	Input	IPC
164.gzip (INT)	input.source 60	0.906
168.wupwise (FP)	wupwise.in	1.392
171.swim (FP)	swim.in	2.267
172.mgrid (FP)	mgrid.in	2.382
173.applu (FP)	applu.in	1.424
175.vpr (INT)	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	0.681
177.mesa (FP)	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm	1.335
179.art (FP)	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	0.299
183.equake (FP)	inp.in	0.494
186.crafty (INT)	crafty.in	1.695
188.ammp (FP)	ammp.in	1.278
197.parser (INT)	ref.in 2.1.dict -batch	0.914
255.vortex (INT)	lendian1.raw	1.880
300.twolf (INT)	ref	0.476
400.perlbench (INT)	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1.545
401.bzip2 (INT)	input.source 280	0.828
403.gcc (INT)	166.i	1.056
416.gamess (FP)	cytosine.2.config	1.879
429.mcf (INT)	inp.in	0.116
433.milc (FP)	su3imp.in	0.458
435.gromacs (FP)	-silent -deffnm gromacs -nice 0	0.595
437.leslie3d (FP)	leslie3d.in	2.205
444.namd (FP)	namd.input	2.4360
445.gobmk (INT)	13x13.tst	0.827
450.soplex (FP)	-s1 -e -m45000 pds-50.mps	0.258
453.povray (FP)	SPEC-benchmark-ref.ini	1.571
456.hmmer (INT)	nph3.hmm	2.362
458.sjeng (INT)	ref.txt	1.421
459.GemsFDTD (FP)	/	2.312
462.libquantum (INT)	1397 8	0.399
464.h264ref (INT)	foreman_ref_encoder_baseline.cfg	1.228
470.lbm (FP)	reference.dat	0.362
471.omnetpp (INT)	omnetpp.ini	0.304
473.astar (INT)	BigLakes2048.cfg	1.252
482.sphinx3 (FP)	ctlfile . args.an4	0.776
483.xalancbmk (INT)	-v t5.xml xalanc.xsl	1.980

Table 2: Benchmarks used for evaluation. Top: CPU2000, Bottom: CPU2006. INT: 18, FP: 18, Total: 36.

6-issue[3] superscalar pipeline. The fetch-to-commit latency is 19 cycles. The in-order frontend and in-order backend are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep frontend (15 cycles) coupled to a shallow backend (4 cycles) to obtain a realistic branch misprediction penalty: 20 cycles minimum. We allow two 16-byte blocks of instructions to be fetched each cycle, potentially over a single taken branch. Table 1 describes the characteristics of the baseline pipeline we use in more details. In particular, the OoO scheduler is dimensioned with a unified centralized 60-entry IQ and a 192-entry ROB on par with the latest commercially available Intel microarchitecture.

As μ-ops are known at *Fetch* in *gem5*, all the widths given in Table 1 are in μ-ops. Independent memory μ-ops (as predicted by the Store Sets predictor [5]) are allowed to issue out-of-order. Entries in the IQ are released upon issue except for those held by memory μ-ops.

The baseline *gem5* simulator does not model speculative scheduling at all, thus the *issue-to-execute* delay is 0 cycle. We refer to this model as the **Baseline_0** configuration ("0" refers to the delay between *Issue* and *Execute*). *Baseline_0* features a dual-ported L1D so as to be able to later isolate the impact of bank conflicts on performance

Bank Conflicts When considering bank conflicts in the L1, we need to implement a mechanism to handle μ-ops that are delayed due to such a conflict.

In Intel Sandy Bridge, *"The L1 DCache maintains requests which cannot be serviced immediately to completion. Some reasons for requests that are delayed: ...loads experiencing bank collisions..."* [10]. Therefore, we use a buffer between the cache and the functional units to queue loads waiting on their bank to do their access. We assume an infinite queue in our experiments.

If two loads are issued in the same cycle (0) and conflict, the second is put in the buffer while the first load begins its access. In the next cycle (1), the second load begins its access. If two loads are also issued during this second cycle, and they both conflict with the load in the buffer, they are in-turn put in the buffer while the second load begins its access. Next cycle (2), the first load of the second issue group will proceed, and the last load will proceed one cycle later (3).

As the cache only supports 2 accesses per cycle, any delay due to an older bank conflict causing a younger load to be put in the buffer will force dependents on the younger load to replay. Such a replay is considered to be due to a bank conflict, albeit indirectly. In the previous example, this will happen if the two younger loads do not conflict with the older load that was queued in cycle (0). Indeed, one of them will still be queued because the cache can only service the older load and one of the younger loads in cycle (1).

[3] On our benchmark set and with our baseline simulator, an 8-issue machine achieves only marginal speedup over this baseline.

Replay Mechanism Our goal is to reduce the overall number of replays. Therefore, the mechanisms we present in Section 5 aim to be agnostic of the replay scheme used to repair schedule misspeculations. Yet, we must still implement one to enforce correctness. For the sake of simplicity, we chose a scheme similar to that used in the Alpha 21264: when a schedule misspeculation is found, all μ-ops between the *Issue* and the *Execute* stage are squashed, and an additional issue cycle is lost. For instance, if issue-to-execute is 4 cycles, then 5 in flight issued groups are lost and no μ-ops are scheduled in the cycle during which the misspeculation is handled. However, in our first experiments, we used a scheme where *all* instructions retained their scheduler entry until they correctly executed. This greatly decreased performance for a 60-entry scheduler. This phenomenon was not observed in previous studies since they either considered a bigger scheduler (128 entries for the 8-wide configuration in [12]), or a less aggressive processor (e.g. 4-wide with a single cycle between issue and execute in [19]). Thus, contention in the IQ was much less pronounced. To alleviate this, we simply use a *Recovery Buffer* similar to the one proposed by Morancho et al. [19] – although more optimistic – to store issued but not yet executed μ-ops.

Specifically, we allow all μ-ops except loads and stores to release their scheduler entry when they issue (speculatively or not). In the meantime, issue groups are placed in the recovery buffer in case they need to be replayed. After a schedule misspeculation, μ-ops are replayed from the recovery buffer, but the scheduler is allowed to fill the holes of issue groups coming from the buffer. The recovery buffer always has priority over the scheduler. We assume that this buffer can handle all μ-ops in flight. Lastly, note that the logic to pick instructions from the buffer only has to select among a single issue group, since instructions were already scheduled the first time they issued.

We refer to *Baseline_** (with "*" the *issue-to-execute* delay) featuring speculative scheduling using this replay mechanism as the **SpecSched_*** model. Unless mentioned otherwise, *SpecSched_* always* schedules load dependents assuming an L1 hit (*Always Hit* policy).

Increasing the Issue-to-Execute Delay In further experiments, we increase the delay between Issue and Execute from 0 cycle to 6 cycles. However, to keep the comparison fair, we keep the branch misprediction penalty constant at 20 cycles. We do so by reducing the number of cycles needed by the frontend to process instructions. For instance, *Baseline_0* has a 15-cycle frontend and a 4-cycle backend while *Baseline_6* has a 9-cycle frontend and a 10-cycle backend, yet in both cases, branches are resolved in cycle 16 (counting from cycle 0), and the penalty is preserved.

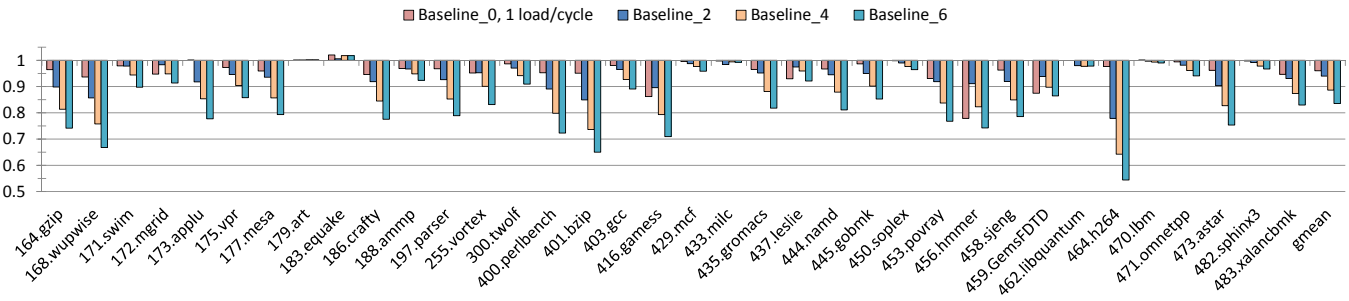


Figure 3: Slowdown due to the increase of the distance between *Issue* and *Execute* (in cycles). No speculative scheduling: dependent on loads are not scheduled until data is available. Baseline is *Baseline_0*.

3.2. Benchmarks

We use a subset[4] of the the SPEC'00 [24] and SPEC'06 [25] suites to evaluate our contributions as we focus on single-thread performance. Specifically, we use 18 integer benchmarks and 18 floating-point programs. Table 2 summarizes said benchmarks as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in each benchmark using Simpoint 3.2 [21]. We simulate the resulting slice in two steps: First, warm up all structures (caches, branch predictor, memory dependency predictor) for 50M instructions, then collect statistics for 100M instructions.

4. Speculative Scheduling

4.1. Overview

As already illustrated in the introduction, speculative scheduling is necessary to enable the execution of chains of dependent μ-ops as early as possible. If D is the *issue-to-execute* delay plus one cycle, in most cases, one can guarantee that the result of instruction I issued at cycle T will be available at cycle $T+D+ExecLat$, provided that its operands are valid. This is not the case for several μ-ops due to circumstances mostly associated with memory accesses.

Nonetheless, efficient speculative scheduling is primordial for modern processors as the *issue-to-execute* delay represents several cycles. This delay includes driving register read commands, register read access time and driving the operands to functional units, including bypass multiplexers By itself, the register file read spans over several cycles [27, 30]. Unfortunately, the *issue-to-execute* delay is not documented on recent processors. On the Alpha 21264 (1996), it is 1 cycle [11] while on the Pentium 4 *Willamette* (2000), the delay is 6 cycles [9]. Intel Haswell (2013) features a 14-cycle pipeline[5] instead of Pentium 4's 20-cycle pipeline. Therefore, unless mentioned otherwise, we will generally consider a 4-cycle *issue-to-execute* delay in this paper.

Fig. 3 illustrates the performance impact of stalling the issuing of μ-ops dependent on a load until the load latency is known, assuming the baseline simulation framework described in Section 3 (in particular, no L1 bank conflicts). It also illustrates the need for the processor to be able to issue more than one load per cycle. Regardless, performance drops very fast as the *issue-to-execute* delay increases; the reason, as pointed out by Borch et al., is that stalling dependents essentially increases the *load-to-use* latency by the distance between *Issue* and *Execute* [4]. Consequently, modern microprocessors should schedule μ-ops as soon as possible – speculatively – to hide the *issue-to-execute* gap. Unfortunately, this introduces the risk of schedule misspeculations.

4.2. Potential Causes of Schedule Misspeculations

In our model, any event that increases the expected execution latency of an instruction will cancel instructions to ensure correctness in the presence of speculative scheduling. Depending on the replay mechanism, this may not be true in general, e.g. *selective replay* only replays dependent instructions that were speculatively issued. If there is no such instruction in flight, no replay is necessary. In the following paragraphs, we detail some well-known events leading to schedule misspeculation, although we do not claim to be exhaustive.

Level 1 Cache Miss Loads are instructions that typically exhibit variable latency. However, said latency cannot generally be known at issue time. Therefore, the scheduler must either schedule dependents hoping for a hit in the L1, or wait for the actual level in which the data resides to be known.

If the scheduler conservatively waits for the memory hierarchy level response to schedule dependents, correctness is preserved, but performance is greatly decreased, as illustrated in Fig. 3 for a baseline 8-wide, 6-issue processor. If, on the contrary, the scheduler aggressively schedules a dependent, performance increases on a hit, but if the load misses, then the dependent must be canceled and replayed as the data is not available when it reaches the execution stage.

It should be noted, however, that in the case of replays due to L1 misses, the performance penalty of predicting a miss but actually hitting in the L1 can be higher than predicting a hit and missing in the L1. In the former case, a dependent will be scheduled at cycle $t = t_{Issue\_Source} + issue\text{-}to\text{-}execute + load\text{-}to\text{-}use$ while it should have been scheduled at cycle

[4] We do not use the whole suites due to some missing system calls/x87 instructions in the *gem5-x86* version we use in this study.

[5] 19 cycles if the micro-operation cache misses.

$t = t_{Issue\_Source} + load\text{-}to\text{-}use$. Therefore, the *load-to-use* of the load is really increased by the *issue-to-execute* delay, and performance is lost if the load is on the critical path. In the latter case (predicted L1 hit, but L1 miss), the dependent will issue too early, but in theory, *no performance is lost on the dependency chain* as the L2 (or higher) response time dominates the replay penalty. In practice, this is not always the case as performance can still be lost due to resource contention or by replaying μ-ops that are independent on the miss. Moreover, energy is wasted anyway since the dependent will issue twice.

Nonetheless, for the dependency chain it affects, the cost of a hit/miss misprediction is *asymmetric*: predicting hit and missing costs energy, while predicting miss and hitting costs performance. Therefore, for high performance, one will tend to favor capturing hits rather than capturing misses.

L1 Cache Bank Conflicts In modern microarchitectures, the processor is able to issue more than one load each cycle [9, 10, 11], meaning that the L1 cache is accessed more than once per cycle. However, to minimize energy and complexity, the data array is banked rather than multiported [10, 22]. That is, the cache can handle several accesses as long as they map to different banks, with potential exceptions.

Two interleaving schemes are possible: set interleaving or word interleaving. Set interleaving allows to interleave both the data array and the tag array. Word interleaving assumes that only the data array is interleaved at the granularity of a quad/double/single-word while the tag array must be replicated or dual-ported. We found that, at equal number of banks, set interleaving performs similarly to a quadword (8-byte) interleaved scheme on our benchmark set. Regardless, the Intel Optimization Guide strongly suggests that Sandy Bridge L1 DCache is organized as an 8-bank, quadword-interleaved structure [10], hence, we use this layout in our experiments.

When two accesses conflict, one has to be delayed, increasing the latency of the corresponding load. As a result, if the scheduler aggressively scheduled a dependent on that load, and although said load hits in the L1, a replay is still triggered. Even worse, should several loads accessing the same bank issue back-to-back for several cycles, then all of them but the first will potentially trigger replays, even if *all of them* hit in the L1. In our framework, two accesses conflict if they access a different set in the cache but map to the same bank. In particular, two accesses to the same set do not conflict, as we consider a cache organization with a Single Line Buffer having two read ports, as described by Rivers et al. [22]. Two concurrent accesses to the same set can take place in a given cycle, although more than two cannot. As a result, the number of bank conflicts is already reduced compared to a banked cache without the Single Line Buffer, which does not play in favor of our mechanism aiming to reduce their number.

Contrarily to replays due to L1 misses, all replays due to L1 bank conflicts have roughly the same cost if no bank conflict prediction technique is present. Indeed, if an L1 miss is assumed but does not take place, and there is a bank conflict during the L1 hit, then dependents pay *issue-to-execute* cycles instead of *bank-conflict-delay* cycles before executing.

However, if the L1 hit had been identified, both energy and performance would have been lost. The former because μ-ops would have been replayed, and the latter because dependents would, again, have paid *issue-to-execute* cycles instead of *bank-conflict delay* cycles before re-executing.

That is, if bank conflicts are not predicted in some fashion, they always cost *issue-to-execute* cycles whether dependents were scheduled assuming an L1 hit or not. In the case where an L1 hit was predicted, additional energy is wasted because dependents are replayed.

To our knowledge, and although Yoaz et al. propose several bank prediction mechanisms [29], bank conflicts are not predicted in current commercially available processors. Naturally, our knowledge is limited as documentation directly related to microarchitectural features is sparse.

Physical Register File (PRF) Bank Conflict Banking the register file is a well known technique to increase the number of physical registers while keeping complexity at bay. If providing each PRF bank with as many read ports as there are sources that can be read each cycle is possible, the interest of banking is that each bank can provision fewer ports to save on energy and delay [27]. In that context, μ-ops of the same issue group can compete for a read port in the same bank, and one of them has to be delayed.

In that event, dependents that were speculatively issued will not execute correctly, and they must be replayed.

PRF Writeback Conflict Depending on the implementation of the bypass network, and due to μ-ops having different execution latencies, it is possible that in a given cycle, more μ-ops than available writeback ports finish executing. In that case, arbitration must take place to ensure all results end up being written back to the register file.

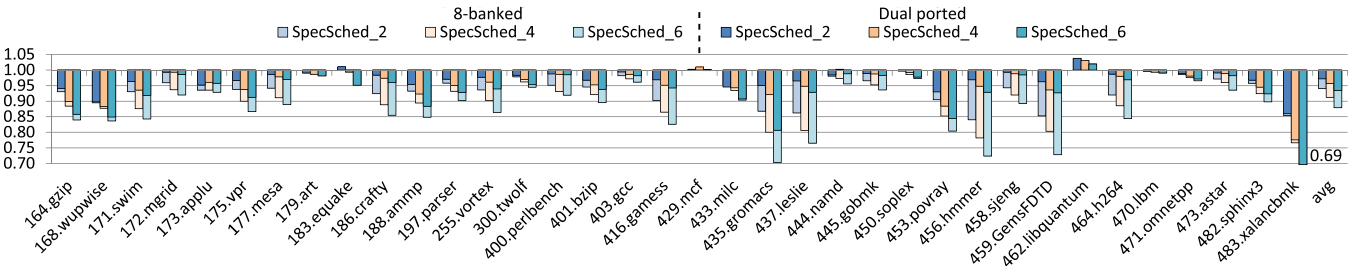
If arbitration is done when registers are read, as in the banked PRF of Tseng et al. [27], then a replay is triggered before the μ-op that would have been delayed is executed.

If, on the contrary, results are buffered at *Writeback* when they cannot be written back due to a lack of write ports [1], then they can still be used by dependents as those buffers are part of the bypass network. In that case, no replay is required.

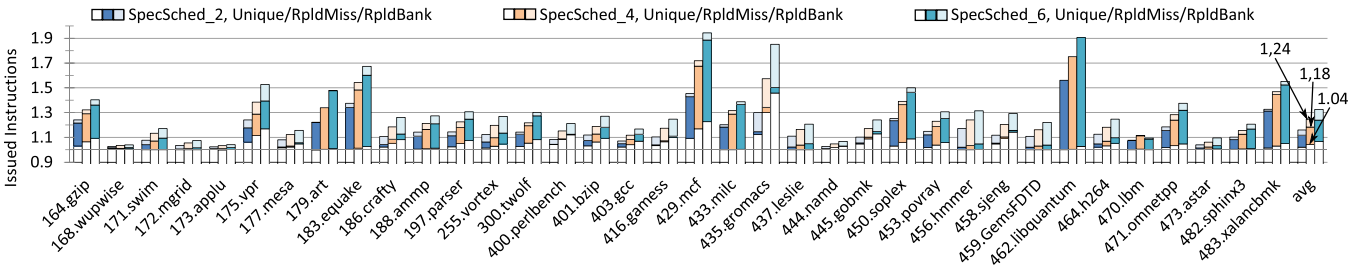
4.3. Summary

There exists many events that can lead to a schedule misspeculation, and some of them depend on design choices made by the architect. In further experiments, we assume a monolithic *Physical Register File* provisioning ports to service all reads and writes taking place in a given cycle. Thus, results cannot contend for a write port in the *Writeback* stage, so we avoid replays due to PRF bank/port conflicts altogether. We only focus on replays caused by L1 misses and L1 bank conflicts.

Nonetheless, among these two causes of replays, L1 bank conflicts may lead to substantial performance degradation



(a) Performance normalized to *Baseline_0*. Darker bars show results for a fully ported L1 while lighter bars show results with a banked L1.



(b) Number of replayed μ-ops and contribution of each replay-triggering event (only for *SpecSched_** with a banked L1), normalized to *Baseline_0*. *Unique*: number of distinct μ-ops issued, *RpldMiss*: μ-ops replayed because of an L1 miss, *RpldBank*: μ-ops replayed because of an L1 bank conflict.

Figure 4: Slowdown due to the increase of the distance between *Issue* and *Execute* (in cycles) on performance and number of replayed μ-ops. Dependents on loads are always scheduled assuming a hit. Baseline is Baseline_0

vs. conservative scheduling (*Baseline_**), as illustrated by Fig. 4 (a). It is paired with Fig. 4 (b) that shows the number of issued μ-ops broken down into three categories: *Unique* (number of distinct μ-ops issued), *RpldMiss* (μ-ops replayed because of an L1 miss) and *RpldBank* (μ-ops replayed because of an L1 bank conflict). In (b), numbers are shown only for *SpecSched_** with a banked L1, but μ-ops replayed because of L1 misses are roughly as numerous with a dual-ported L1D. Note that as *issue-to-execute* delay increases, so does the number of distinct μ-ops issued. This is because although the minimum *fetch-to-branch resolution* delay is not increased, the average delay is, due to reduced performance. Moreover, the *issue-to-branch resolution* increases, allowing more μ-ops to be issued on the wrong path after a branch issued.

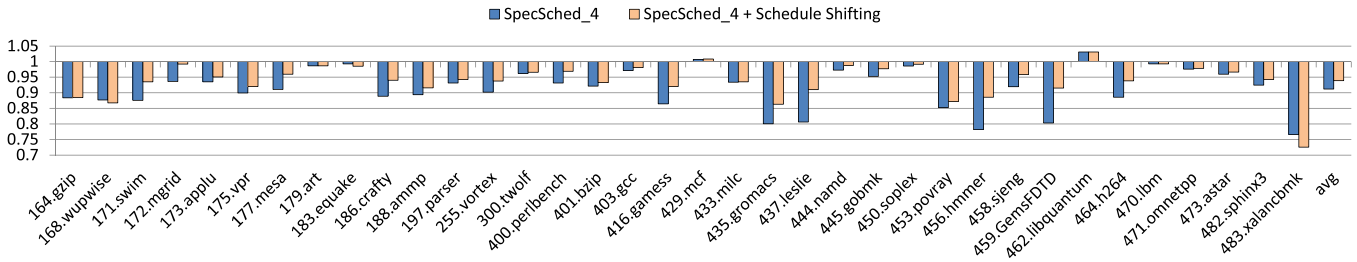
For the darker set of bars of (a), a fully dual-ported L1 is assumed and in general, performance is increased vs. the conservative scheduling of Fig. 3. However, (b) suggests that in many cases, L1 misses force many μ-ops to replay, e.g *art, equake, mcf, milc, gromacs, soplex, libquantum, omnetpp* and *xalancbmk*. This does not lead to a performance decrease, except for *xalancbmk*. In that case, the base IPC is 1.98 in *Baseline_0* but the L1 miss rate is 46%. Therefore, many replays take place in *SpecSched_**, and since ILP is high, many independent μ-ops are also replayed. Other high IPC benchmarks (*swim, mgrid, namd, hmmer* and *gemsFDTD*) are all over 2 IPC but have a much lower L1 miss rate (2% at most), and are therefore not subject to this phenomenon.

Conversely, *libquantum* is slightly sped up when using speculative scheduling. This is because most accesses are actually L1 misses, hence, in the baseline case, the scheduler is full very often. With our recovery buffer scheme, even if most accesses are misses, dependents are speculatively removed from the scheduler the first time they issue, hence, the frontend is able to bring slightly more instructions in the window. Because ILP is low in *libquantum*, instructions from the recovery buffer and from the scheduler rarely contend for resources.

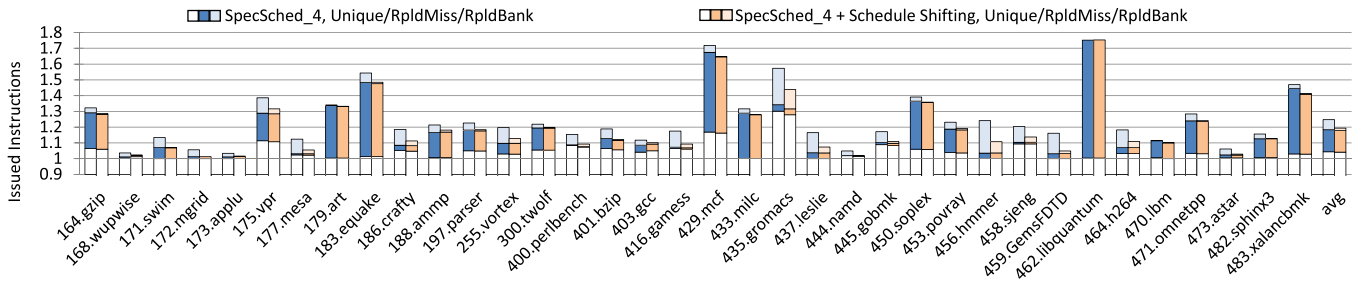
Overall, this suggests that speculative scheduling mitigates the performance loss due to the *issue-to-execute* delay, although performance is generally lower than the ideal *Baseline_0* due to replayed μ-ops.

The set of lighter bars takes bank conflicts into account (8 8-byte banks, quadword interleaved). Several benchmarks (e.g. *swim, crafty, gamess, gromacs, leslie, hmmer, GemsFDTD* and *h264*) lose more than 5% performance to replays due to bank conflicts when the *issue-to-execute* delay is 4 cycles. This correlates well with Fig. 4 (b) where benchmarks losing the most performance when the cache is banked are actually those having the biggest proportion of μ-ops replayed due to L1 bank conflicts, on average.

As a result, bank conflicts should be taken into account when designing a processor with speculative scheduling, and a smarter policy than *Always Hit* for hit/miss prediction would be beneficial in some cases (e.g. *xalancbmk*) [9, 11]. Moreover, this supports our hypothesis that an isolated replay due to an L1 miss costs less performance (at least with our recovery scheme) than a replay due to an L1 bank conflict. Indeed, if the average performance loss due to L1 misses is roughly similar to the performance loss due to L1 bank conflicts, roughly twice as many μ-ops are replayed because of L1 misses in *SpecSched_4*.



(a) Performance of *SpecSched_4* with *schedule shifting* normalized to *Baseline_0*.



(b) Number of replayed μ-ops and contribution of each replay-triggering event, normalized to *Baseline_0*.

Figure 5: Impact of schedule shifting on performance and replayed μ-ops. Baseline is *Baseline_0*.

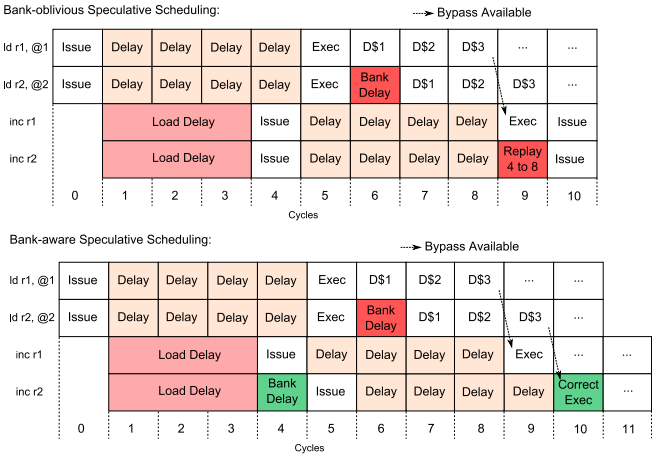


Figure 6: Execution of two loads conflicting in the L1 followed by two dependent μ-ops. Top: A replay takes place because the second load returns late. Bottom: The dependent on the second load was scheduled *bank-delay* late, avoiding a replay.

Regardless, our goal is to provide simple ways to reduce the number of replays due to both of these events. These mechanisms are orthogonal to the PRF layout, and could therefore be combined to any mechanism aiming to limit the replays due to PRF bank/port conflicts.

5. Experimental Results

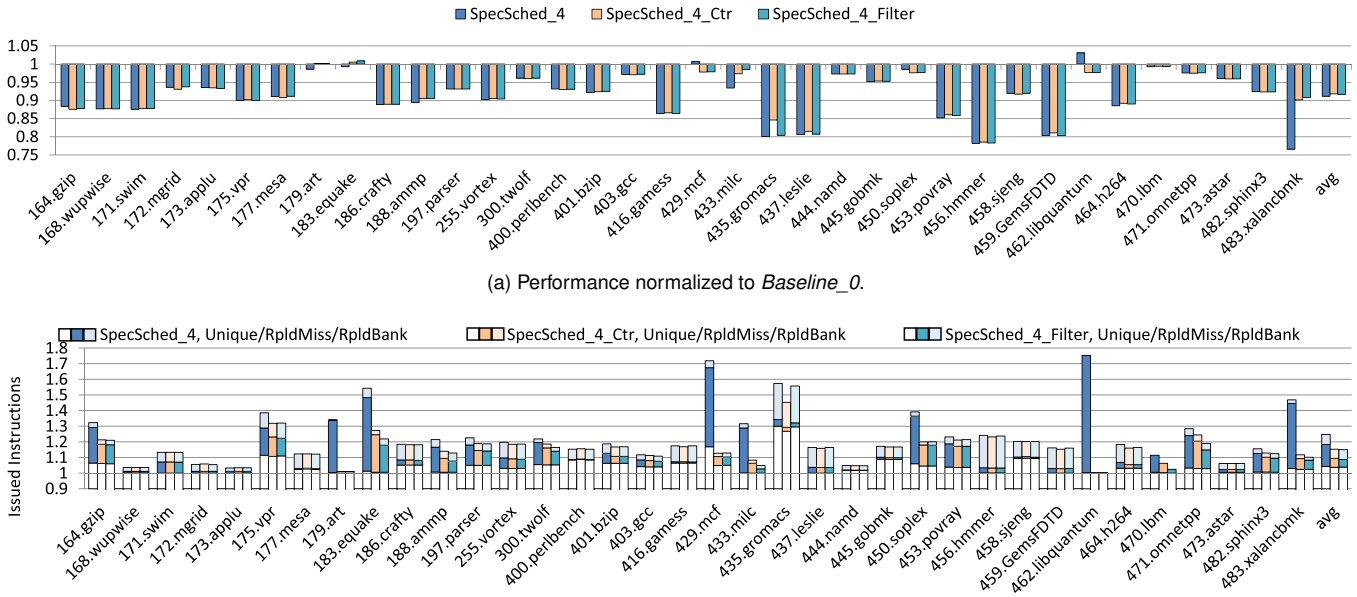
In this section, **all experiments consider a banked L1 cache**, unless mentioned otherwise. Moreover, to ease figure comparison, we always use *Baseline_0* with a dual-ported L1D as the baseline as it represents the ideal performance in this context. It follows that our mechanisms aim to increase performance over *SpecSched_** with a banked L1D to bridge the gap between the latter and *Baseline_0*. They do not aim for a speedup over *Baseline_0*. Lastly, when averaging speedups, the geometric mean is used.

5.1. Schedule Shifting

The simplest solution to avoid most bank conflicts is as follows. Although we issue two loads in the same cycle, we speculatively wake up dependents on the second one with a latency increased by one. In other words, we always expect pairs of loads to conflict in the L1. We refer to this policy as *schedule shifting*. An example is given in Fig. 6 where two conflicting loads are issued in the same cycle, followed by their dependents after *load-to-use* cycles. In particular, the top part shows that without *schedule shifting*, μ-ops must be replayed when a bank conflict takes place (even the dependent on the first load, *inc r1*, must be replayed without *selective replay*). Conversely, in the bottom part, the dependent on the second load is issued one cycle later, and no replay has to take place.

This has three main drawbacks. First, if the pair of loads does not conflict, then the load-to-use latency of the second load is increased by one cycle. Second, this does not prevent bank conflicts from happening. Indeed, it is possible that the second load of a given issue group will conflict with the first load of the next issue group. Third, in cases where two loads issue the same cycle and both miss, this will trigger two replays instead of one because of the extra cycle the second load was granted to execute. However, it is very simple to implement, and it greatly reduces the number of L1 bank conflicts.

Results Fig. 5 (a) shows that part of the performance lost when we introduced L1 bank conflicts is recovered. Performance is improved by 2.9% over *SpecSched_4* on average,



(a) Performance normalized to *Baseline_0*.

(b) Number of replayed μ-ops and contribution of each replay-triggering event, normalized to *Baseline_0*.

Figure 7: Impact of filtering hits and misses with different mechanisms on performance and number of replayed μ-ops for *SpecSched_4* vs. *Baseline_0*

while it would be improved by 4.8% if the cache were dual-ported (see the darker bars of Fig. 4 (a)). Then, Fig. 5 (b) tells us that the number of μ-ops replayed due to L1 bank conflicts is greatly reduced, by 74.8% on average. That is, we are able to greatly mitigate the impact of L1 bank conflicts on performance by using a very simple mechanism.

Through enhancing performance, *schedule shifting* also marginally reduces the number of distinct issued μ-ops (*Unique*) in some benchmarks (e.g. *perlbench, mcf* and *gromacs*). The reason is that through always delaying dependents on the second load issued each cycle, fewer μ-ops are issued on the wrong path.

5.2. Using Hit/Miss Filtering to Limit Replays

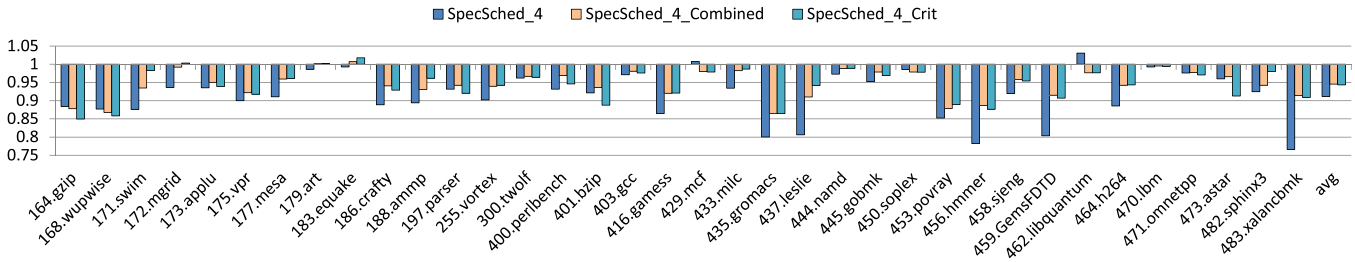
Using a Global Counter Since L1 misses tend to be grouped in time, we consider a global counter to drive the aggressive scheduling of μ-ops. In particular, we use the exact same scheme as the Alpha 21264 [11]: the most significant bit of a 4-bit counter tells if a load should speculatively wake up its dependents or not. The counter is decremented by two on cycles where an L1 miss takes place, and incremented by one otherwise. This configuration is referred to as **SpecSched_4_Ctr**.

Per-Instruction Filter There already exist propositions to predict L1 hit/miss [29], and the Pentium 4 also uses some form of – undisclosed – hit/miss prediction [9]. However, complex schemes may not be cost-efficient, especially from a performance standpoint. Therefore, the second approach we use to reduce replays due to L1 misses is a simple per-instruction filter. In particular, we consider a 2K-entry, direct-mapped array of 2-bit saturating counters that are incremented on a hit and decremented on a miss. In each entry, an additional bit allows to silence the counter if it goes from a saturated state to a transient state (e.g. from 0 to 1 after a hit). If a counter is not silenced, this means that the μ-op has always hit – resp. missed – until now. If the counter is silenced, this means that the μ-op does not *always hit* – resp. *always miss* – and we let the 4-bit counter of the previous policy decide if dependents should be scheduled or not.

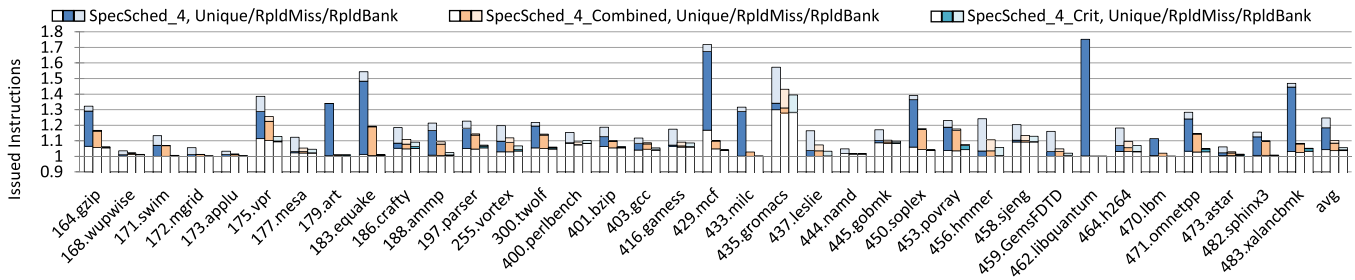
To avoid counters being *silenced* forever, we reset the whole array of *silent* bits every 10K committed loads. Counters are not updated while they are silenced. This configuration is referred to as **SpecSched_4_Filter**. Lastly, note that this filter is not on the critical path and is updated at commit time.

The rationale of using a bit to silence the counter – as opposed to simply using the most significant bit to predict hit or miss – is that the behavior of some loads is strongly dependent on the behavior of very recent dynamic loads. Without the silencing bit, an entry may saturate to hit often while the behavior of the load actually follows that of a recent dynamic load, which sometimes misses. In that case, a replay will be triggered every time said load misses, while using the global counter would have prevented it. In our experiments, we found that using a silencing bit performs better than regular per-entry counters, thus we only report numbers for the former scheme.

Results As depicted in Fig. 7 (a), using a single counter has mostly no impact on performance except in *mcf* and *libquantum*. In those cases, performance decreases as the counter unnecessarily categorize μ-ops that hit as misses because several misses have been observed recently. Yet, this scheme allows to reduce the overall number of μ-ops replayed due to L1 misses by 59.3% on average, and by respectively 88.7%



(a) Performance normalized to *Baseline_0*.



(b) Number of replayed μ-ops and contribution of each replay-triggering event, normalized to *Baseline_0*.

Figure 8: Performance and number of replayed μ-ops of *SpecSched_4_Combined* and *SpecSched_4_Crit* vs. *Baseline_0*.

and more than 99% for *mcf* and *libquantum*. In the latter case, most of the accesses miss in the L1, hence the *Always Hit* policy is clearly not adapted. The overall number of replayed μ-ops is reduced by 44.7% vs. *SpecSched_4*, on average.

By allowing some loads not to wake up their dependents aggressively, performance is substantially improved in *xalancbmk*. The reason is that *xalancbmk* has high IPC but suffers from many L1 misses, hence replays.

Similarly to the previous scheme, performance remains mostly untouched when using both global counter and filter, as shown in Fig. 7 (a). However, using only 768 Bytes of storage, we manage to reduce the number of μ-ops replayed due to L1 misses by 65.0% on average. The overall number of replayed μ-ops is reduced by 45.4% on average. As a result, depending on the cost of the replay policy that is implemented, spending transistors on that filter can be a valid tradeoff. Moreover, if the performance of *gromacs* decreases by using the filter and the global counter instead of the global counter only, the number of instructions replayed because of L1 misses actually slightly decreases. Unfortunately, this in turn allows loads that access the same bank to be issued more often, leading to an increase in the number of instructions replayed due to bank conflicts.

Lastly, in both cases, the number of *Unique* μ-ops issued in some benchmarks (e.g. *vpr, mcf, soplex* and *xalancbmk*) is diminished by forbidding the scheduler from issuing dependents μ-ops as soon as possible. This is due to the fact that fewer μ-ops may be in flight to *Execute* on the wrong path.

5.3. Combining Both Mechanisms

In previous experiments, we focused on distinct mechanisms aiming to reduce either the number of replays due to L1 misses or to L1 bank conflict. In this Section, we combine both mechanisms (**SpecSched_4_Combined** configuration). We also show that for a certain criterion of *criticality*, we can keep most of the performance brought by speculative scheduling while further reducing the number of replays.

Criticality Estimation Although there has been several complex schemes proposed to estimate the criticality of a μ-op, [7, 8, 28], we consider a simple one in these experiments, as a proof of concept. We mark a μ-op critical if it was at the head of the ROB when it completed during previous executions [7, 28]. We refer to this configuration as **SpecSched_4_Crit**. Then, unless it is a sure hit as predicted by the hit/miss filter, we do not wake up μ-ops dependent on a non-critical load speculatively. Critical loads that are not filtered out by the hit/miss filter are processed using the global counter.

We use an 8K-entry direct-mapped table containing small signed counters (4-bit in our experiments). A counter is incremented if a μ-op has been found critical during the last execution, and decremented otherwise. The prediction is then given by the most significant bit. This structure is not on the critical path, and it is only updated at retire time.

Results Fig. 8 (a) and (b) report numbers for *SpecSched_4_Combined* and *SpecSched_4_Crit*.

First, although a slight decrease in performance can be observed for *gzip* and *wupwise*, *SpecSched_4_Combined* generally performs better than *SpecSched_4*, by 3.7% on average. Moreover, as depicted in Fig. 8 (b), the overall number of replayed μ-ops is greatly decreased, by 68.2% on average.

Second, if using criticality to drive the scheduler costs performance in some cases (e.g. *gzip, bzip, hmmer, astar*), average performance still increases by 3.4% over *SpecSched_4*, which

is comparable to *SpecSched_Combined*. However, the interest of this scheme is in the reduction of replayed μ-ops, where it performs well: 90.6% fewer μ-ops replayed, on average.

As a result, we are able to recover most of the speedup brought by speculative scheduling without L1 bank conflicts reported in Fig. 4 (a) (3.47%/3.4% for *SpecSched_4_Combined/Crit* vs. 4.8% for *SpecSched_4* with a dual-ported cache) while actually having a banked L1 cache. In the meantime, the number of replays due to the variable latency of loads is reduced by an order of magnitude.

By lack of space, we do not report results for *SpecSched_2_Crit* and *SpecSched_6_Crit*. However, we found that the decrease in number of replayed instructions vs. the equivalent *SpecSched_** configuration was roughly constant, around 90% in both cases. These improvements lead to a reduction in the total number of issued instructions of 11.2% for *SpecSched_2_Crit* over *SpecSched_2* and 18.7% for *SpecSched_6_Crit* overs *SpecSched_6*, on average. As for *SpecSched_4_Crit*, speedup is also observed: 2.3% and 4.8% respectively. Therefore our contributions appear efficient regardless of the *issue-to-execute* delay.

Third and last, we benefit from the reduction of the *Unique* category brought by *schedule shifting* and hit/miss filtering. If on average, only 0.7% fewer *Unique* μ-ops are issued by both *SpecSched_4_Combined* and *SpecSched_4_Crit*, in some benchmarks such as *mcf*, the reduction can grow to respectively 10.4% and 11.2%.

6. Conclusion

To get the best performance out of a modern microprocessor, speculative scheduling is necessary under penalty of increasing the load-to-use delay by several cycles. As a result, it is very likely that commercially available processors speculatively schedule dependents on variable latency μ-ops and feature a replay mechanism to handle misspeculations. Unfortunately, the currently implemented replay as well as replay-avoiding mechanisms (e.g. hit/miss prediction) are unknown as many architectural features remain undocumented.

Yet, if the precise implementation of the replay mechanism impacts performance and energy, it addresses effects and not causes. In this work, we focus on two types of events leading to replays – L1 misses and L1 bank conflicts – and devise simple mechanisms to greatly mitigate their impact. We also argue that isolated replays do not have the same cost depending on which event triggered them. In particular, wrongly assuming an L1 hit has low performance cost as long as no independent μ-ops are canceled (e.g. for moderate IPC benchmarks), but replays due to L1 bank conflicts always cost performance.

To that extent, we propose several mechanisms to reduce both types of replays. First, we propose *schedule shifting* to reduce the number of replays due to L1 bank conflicts. In particular, by always waking dependents on the second load of an issue-group with a one-cycle delay, we achieve a 2.9% speedup on average while reducing the number of replayed μ-ops due to bank conflicts by 74.8% on average.

Second, we study the impact of using a 4-bit global counter to drive the aggressive scheduling of load dependents. We combine it to a very simple filter requiring less than 1KB of storage, and show that replays due to L1 misses can be reduced by 65.0% on average. While this generally have little impact on performance, in benchmarks where IPC is high but hitrate is low (e.g. *xalancbmk*), this mechanism mitigates the loss of performance due to the high number of canceled μ-ops.

Finally, by combining both schemes and taking into account the criticality of a load using its position in the ROB when it completed as a criterion, we are able to actually remove the majority of the replays observed in *SpecSched_4*: respectively 68.2% and 90.6% for *SpecSched_4_Combined* and *SpecSched_4_Crit*. This leads to a reduction in the number of issued μ-ops of 11.6% and 13.4%, respectively. In the meantime, performance is still slightly increased by 3.7% and 3.4% respectively, over *SpecSched_4*.

That is, by using simple hardware that is outside the critical path, we are able to increase the efficiency of the pipeline by issuing roughly the same number of μ-ops as a pipeline without speculative scheduling – *Baseline_4* issues 15.6% fewer μ-ops than *SpecSched_4* – while slightly increasing performance over a baseline speculative scheduling scheme.

Acknowledgments

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175.

References

[1] R. Balasubramonian, H. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the International Symposium on Microarchitecture*, 2001, pp. 237–248.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proceedings of the International Symposium on Computer Architecture*, 1999, pp. 54–63.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[4] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, Feb 2002, pp. 299–310.

[5] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the International Symposium on Computer Architecture*, 1998, pp. 142–153.

[6] D. Ernst, A. Hamel, and T. Austin, "Cyclone: a broadcast-free dynamic instruction scheduler with selective replay," in *Proceedings of the International Symposium on Computer Architecture*, June 2003, pp. 253–262.

[7] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the International Symposium on Computer Architecture*, 2001, pp. 74–85.

[8] B. R. Fisk and R. I. Bahar, "The non-critical buffer: Using load latency tolerance to improve data cache efficiency," in *International Conference on Computer Design*, 1999, pp. 538–545.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, vol. 1, p. 2001, 2001.

[10] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[11] R. E. Kessler, E. J. Mclellan, and D. A. Webb, "The Alpha 21264 microprocessor architecture," in *Proceedings of the International Conference on Computer Design*, 1998, pp. 90–95.

[12] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2004, pp. 198–.

[13] Y. Liu, A. Shayesteh, G. Memik, and G. Reinman, "Scaling the issue window with look-ahead latency prediction," in *Proceedings of the International Conference on Supercomputing*, 2004, pp. 217–226.

[14] G. Memik, G. Reinman, and W. H. Mangione-Smith, "Precise instruction scheduling," *Journal of Instruction-Level Parallelism*, vol. 7, pp. 1–29, 2005.

[15] A. Merchant, D. Boggs, and D. Sager, "Processor with a replay system that includes a replay queue for improved throughput," 2007, US Patent 7,200,737.

[16] A. Merchant and D. Sager, "Computer processor having a checker," 2001, US Patent 6,212,626.

[17] A. Merchant, D. Sager, D. Boggs, and M. Upton, "Computer processor with a replay system having a plurality of checkers," 2000, US Patent 6,094,717.

[18] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2001, pp. 27–36.

[19] E. Morancho, J. M. Llabería, and À. Olivé, "Recovery mechanism for latency misprediction," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp. 118–128.

[20] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Proceedings of the International Symposium on Computer Architecture*, 1997, pp. 206–218.

[21] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 244–.

[22] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, "On high-bandwidth data cache design for multi-issue processors," in *Proceedings of the International Symposium on Microarchitecture*, 1997, pp. 46–56.

[23] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

[24] Standard Performance Evaluation Corporation. CPU2000. Available: http://www.spec.org/cpu2000/

[25] Standard Performance Evaluation Corporation. CPU2006. Available: http://www.spec.org/cpu2006/

[26] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *Proceedings of the International Symposium on Microarchitecture*, 2000, pp. 57–66.

[27] J. H. Tseng and K. Asanović, "Banked multiported register files for high-frequency superscalar microprocessors," in *Proceedings of the International Symposium on Computer Architecture*, 2003, pp. 62–71.

[28] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 104–113.

[29] A. Yoaz, R. Erez, M.and Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the International Symposium on Computer Architecture*, vol. 27, no. 2, 1999, pp. 42–53.

[30] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1998, pp. 305–310.